Factorization of a 768-bit RSA modulus

Thorsten Kleinjung[1], Kazumaro Aoki[2], Jens Franke[3], Arjen K. Lenstra[1], Emmanuel Thomé[4], Joppe W. Bos[1], Pierrick Gaudry[4], Alexander Kruppa[4], Peter L. Montgomery[5,6], Dag Arne Osvik[1], Herman te Riele[6], Andrey Timofeev[6], and Paul Zimmermann[4]

[1] EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland
[2] NTT, 3-9-11 Midori-cho, Musashino-shi, Tokyo, 180-8585 Japan
[3] University of Bonn, Dept. of Math., Beringstraße 1, D-53115 Bonn, Germany
[4] INRIA CNRS LORIA, Équipe CARAMEL - bâtiment A, 615 rue du jardin botanique, F-54602 Villers-lès-Nancy Cedex, France
[5] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
[6] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract. This paper reports on the factorization of the 768-bit number RSA-768 by the number field sieve factoring method and discusses some implications for RSA.
Keywords: RSA, number field sieve

1 Introduction

On December 12, 2009, we factored the 768-bit, 232-digit number RSA-768 by the number field sieve (NFS, [19]). RSA-768 is a representative 768-bit RSA modulus [34], taken from the RSA Challenge list [35]. Our result is a new record for factoring general integers. Factoring a 1024-bit RSA modulus would be about a thousand times harder and a 512-bit one was several thousands times easier. Because the factorization of a 512-bit RSA modulus [7] was first reported in 1999, it is not unreasonable to expect that 1024-bit RSA moduli can be factored well within the next decade by a similar academic effort. Thus, it would be prudent to phase out usage of 1024-bit RSA within the next three to four years.

The previous NFS record was the May 9, 2005, factorization of the 663-bit, 200-digit number RSA-200 [4]. NFS records must not be confused with *special* NFS (SNFS) records. The current SNSF record is the May 21, 2007, factorization of the 1039-bit number $2^{1039} - 1$ [2]. Although much bigger than RSA-768, its special form made $2^{1039} - 1$ an order of magnitude easier to factor.

The new NFS record required the following effort. We spent half a year on 80 processors on polynomial selection. This was about 3% of the main task, the *sieving*, which took almost two years on many hundreds of machines. On a single core 2.2 GHz AMD Opteron processor with 2 GB RAM, sieving would have taken about fifteen hundred years. We did about twice the sieving strictly necessary, to make the most cumbersome step, the *matrix step*, more manageable. Preparing the sieving data for the matrix step took a couple of weeks on a few processors. The final step after the matrix step took less than half a day of computing.

Sieving is a laid back process that, once running, does not require much care beyond occasionally restarting a machine. The matrix step is more subtle. A slight disturbance easily causes major trouble, in particular if the problem stretches the available resources. *Oversieving* led to a matrix that could be handled relatively smoothly. More importantly, the extra sieving data allow experiments aimed at getting a better understanding of the relation between sieving and matrix efforts and the effect on NFS feasibility and overall performance. That work is in progress. All in all, the extra sieving time was well spent.

In [2] the block Wiedemann algorithm [9] was used, making it possible to process the matrix on disjoint clusters. Larger problems (such as 1024-bit moduli) require wider parallelization. Here we solve some of the challenges by dividing the workload in a more flexible manner. As a result a matrix nine times harder than in [2] was solved in less than four months, on clusters in three countries. Larger matrices are within reach and there can be little doubt about the feasibility by the year 2020 of the matrix for a 1024-bit modulus. We are studying if our current matrix can be handled by block Lanczos [8] on a single cluster.

The steps taken to factor RSA-768 are described in Section 2. The factors are given in Section 2.5. Implications for moduli larger than RSA-768 are briefly discussed in Section 3. Appendix A presents the sieving approach that we used, and Appendix B describes a new twist of the block Wiedemann algorithm that makes it easier to share large calculations among different parties.

2 Factoring RSA-768

2.1 Factoring using the Morrison-Brillhart approach

The *congruence of squares* method factors a composite n by writing it as $\gcd(x-y,n)\cdot\gcd(x+y,n)$ for integers x,y with $x^2\equiv y^2 \bmod n$: for random such pairs (x,y) the probability of success is at least $\frac{1}{2}$. We explain the Morrison-Brillhart approach [25] to solve $x^2\equiv y^2 \bmod n$ and, roughly, how NFS works.

A non-zero integer u is b-smooth if the prime factors of $|u|$ are at most b. Each b-smooth integer u corresponds to the $(\pi(b)+1)$-dimensional integer vector $\mathfrak{v}(u)$ of exponents of the primes $\leq b$ in its factorization, where $\pi(b)$ is the number of primes $\leq b$ and the "+1" accounts for the sign. The *factor base* consists of the primes at most equal to the smoothness bound.

Let n be a composite integer, b a smoothness bound, and t a positive integer. Let V be a set of $\pi(b)+1+t$ integers v for which the least absolute remainders $r(v)=v^2 \bmod n$ are b-smooth. Because the $(\pi(b)+1)$-dimensional vectors $\mathfrak{v}(r(v))$ are linearly dependent, at least t independent subsets $T\subset V$ can be found using linear algebra such that $\sum_{v\in T}\mathfrak{v}(r(v))$ is an all-even vector. Thus, each T leads to a solution $x=\prod_{v\in T} v$ and $y=\sqrt{\prod_{v\in T} r(v)}$ to $x^2\equiv y^2 \bmod n$. Overall, this *combining of congruences* results in t chances of at least $\frac{1}{2}$ to factor n.

In Dixon's random squares method [11] the set V is generated by randomly selecting integers v until enough have been found for which $r(v)$ is smooth. The expected runtime can be proved rigorously. With quadratic residues $r(v)$ of

order n, however, the method is not practical: earlier, Morrison and Brillhart [25] had already shown how to use continued fractions to generate quadratic residues of order $n^{1/2}$. The much higher smoothness probabilities make their method much faster than Dixon's, despite the lack of a formal proof. Schroeppel with his *linear sieve* was the first, in about 1976, to combine similarly high smoothness probabilities with fast *sieving*-based smoothness detection [31, Section 6] and to analyze the resulting heuristic expected runtime [17, Section 4.2.6]. A variation led to Pomerance's *quadratic sieve* [31,32]. Factoring methods of this sort that rely on smoothness of residues of order $n^{\theta(1)}$ have expected runtimes of the form

$$e^{(c+o(1))(\ln n)^{1/2}(\ln\ln n)^{1/2}} \quad (\text{for } n \to \infty)$$

for positive constants c. The number field sieve [19] was the first, and so far the only, practical factoring method to break through the barrier of the $\ln n$-exponent of $\frac{1}{2}$. It uses more contrived congruences that involve smoothness of numbers of order $n^{o(1)}$, for $n \to \infty$, that can, as usual, be combined into a congruence of squares $x^2 \equiv y^2 \bmod n$. NFS factors a composite integer n in heuristic expected time

$$e^{((64/9)^{1/3}+o(1))(\ln n)^{1/3}(\ln\ln n)^{2/3}} \quad (\text{for } n \to \infty).$$

It is currently the best algorithm to factor numbers without special properties, such as RSA-768, a 768-bit, 232-digit RSA modulus taken from [35]:

123018668453011775513049495838496272077285356959533479219732245215172640050726
365751874520219978646938995647494277406384592519255732630345373154826850791702
612214291346167042921431160222124047927473779408066535141959745985690214341 3.

Similar to Schroeppel's linear sieve, the most important steps of NFS are *sieving* and the *matrix step*. In the former *relations* are collected, congruences involving smooth values similar to the smooth $r(v)$-values above. In the latter linear dependencies are found among the exponent vectors of the smooth values. NFS requires two non-trivial additional steps: a pre-processing *polynomial selection* step before the sieving can start, and a post-processing *square root step* to convert the linear dependencies into congruences of squares. A rough operational description of these steps as applied to RSA-768 is given below. For an explanation why these steps work, we refer to the many expositions on NFS [19,20,33].

2.2 Polynomial selection

With n the integer to be factored, let $f_1(X), f_2(X) \in \mathbf{Z}[X]$ be two irreducible integer polynomials of degrees d_1 and d_2, respectively, with a common root m modulo n, i.e., $f_1(m) \equiv f_2(m) \equiv 0 \bmod n$. For simplicity we assume that f_1 and f_2 are monic, even though the actual f_1 and f_2 are not. With $v_k(a,b) = b^{d_k} f_k(a/b) \in \mathbf{Z}$ $(k = 1, 2)$, relations are coprime pairs of integers (a, b) with $b > 0$ such that $v_1(a, b)$ and $v_2(a, b)$ are simultaneously smooth, $v_1(a, b)$ with respect to some b_1 and $v_2(a, b)$ with respect to some b_2. Sufficiently many more than $\pi(b_1)+\pi(b_2)+2$ relations lead to enough chances to factor n, as sketched below.

Let $\mathbf{Q}(\alpha_k) = \mathbf{Q}[X]/(f_k(X))$ for $k = 1, 2$ be two algebraic number fields. The elements $a - b\alpha_k \in \mathbf{Z}[\alpha_k]$ have *norm* $v_k(a, b)$ and belong to the first degree prime

ideals in $\mathbf{Q}(\alpha_k)$ of (prime) norms equal to the prime factors of $v_k(a,b)$. These prime ideals in $\mathbf{Q}(\alpha_k)$ correspond bijectively to the pairs $(p, r \bmod p)$ where p is prime and $f_k(r) \equiv 0 \bmod p$: excluding factors of f_k's discriminant, such a first degree prime ideal has norm p and is generated by p and $r - \alpha_k$.

Because $f_k(m) \equiv 0 \bmod n$, the two natural ring homomorphisms $\phi_k : \mathbf{Z}[\alpha_k] \to \mathbf{Z}/n\mathbf{Z}$ for $k = 1, 2$ map $\sum_{i=0}^{d_k-1} a_i \alpha_k^i$ to $\sum_{i=0}^{d_k-1} a_i m^i \bmod n$ and $\phi_1(a - b\alpha_1) \equiv \phi_2(a - b\alpha_2) \bmod n$. Linear dependencies modulo 2 among the exponent-vectors of the primes in the b_1-smooth $v_1(a,b)$, b_2-smooth $v_2(a,b)$ pairs lead to subsets T such that $\prod_{(a,b)\in T}(a - b\alpha_k)$ is a square σ_k in $\mathbf{Q}(\alpha_k)$, for $k = 1, 2$. With $\phi_1(\sigma_1) \equiv \phi_2(\sigma_2) \bmod n$ it then remains to compute square roots $\tau_k = \sqrt{\sigma_k} \in \mathbf{Q}(\alpha_k)$ for $k = 1, 2$ to find a solution $x = \phi_1(\tau_1)$ and $y = \phi_2(\tau_2)$ to $x^2 \equiv y^2 \bmod n$.

It is easy to find f_1 and f_2 so that numbers of order $n^{o(1)}$, for $n \to \infty$, must be smooth. Let $d_1 \in \mathbf{N}$ be of order $(\frac{3 \ln n}{\ln \ln n})^{1/3}$, let $d_2 = 1$, let m be an integer slightly smaller than n^{1/d_1}, and write n in radix m as $n = \sum_{i=0}^{d_1} n_i m^i$ with $0 \le n_i < m$. Then $f_1(X) = \sum_{i=0}^{d_1} n_i X^i$ and $f_2(X) = X - m$ have common root m modulo n, the coefficients are $n^{o(1)}$ for $n \to \infty$, and the values a, b that suffice to generate enough relations are small enough to keep $b^{d_1} f_1(a/b)$ and $b^{d_2} f_2(a/b)$ of order $n^{o(1)}$ as well. Finally, if f_1 is not irreducible, it can be used to directly factor n or, if that fails, one of its factors can be used instead of f_1. If $d_1 > 1$ and $d_2 = 1$ we refer to "$k = 1$" as the *algebraic side* and "$k = 2$" as the *rational side*. With $d_2 = 1$ the algebraic number field $\mathbf{Q}(\alpha_2)$ is simply $\mathbf{Q}$, the first degree prime ideals in $\mathbf{Q}$ are the regular primes and, with $f_2(X) = X - m$, the element $a - b\alpha_2$ of $\mathbf{Z}[\alpha_2]$ is $a - bm = v_2(a,b) \in \mathbf{Z}$ with $\phi_2(a - b\alpha_2) = a - bm \bmod n$.

Although with these polynomials NFS achieves its asymptotic runtime, there is a lot of freedom in the choices of m, f_1, and f_2. Exploiting this involves extensive searches, comparing choices based on smoothness probabilities, and thus with respect to coefficient size, number of real roots and roots modulo small primes, smoothness properties of leading coefficients, and sieving experiments. How the search is best conducted is the subject of active research; current approaches are guided by experience, helped by luck, and profit from patience.

One method is known that produces two good polynomials of degrees greater than one (namely, twice degree two [5]). Its results are not competitive with the current best $d_1 > 1$, $d_2 = 1$ methods which are all based on refinements [15] of the approach from [24,26] as summarized in [7, Section 3.1]. A search of three months on a cluster of 80 Opteron cores (i.e., $\frac{3}{12} \cdot 80 = 20$ core years), conducted at BSI in 2005 already and thus not including the idea from [16], produced three pairs of polynomials of comparable quality. We used

$$\begin{aligned}
f_1(X) &= 265482057982680X^6 \\
&\quad + 1276509360768321888X^5 \\
&\quad - 5006815697800138351796828X^4 \\
&\quad - 4647785447172785427172677450X^3 \\
&\quad + 6525437261935989397109667371894785X^2 \\
&\quad - 1818577935208859435672601886243480305 4X \\
&\quad - 2775652667915438819952161997138011033431 20, \\
f_2(X) &= 3466100355049250185144582 9X - 1291187456580021223163547791574810881.
\end{aligned}$$

The leading coefficients factor as $2^3 \cdot 3^2 \cdot 5 \cdot 7^2 \cdot 11 \cdot 17 \cdot 23 \cdot 31 \cdot 112\,877$ and $13 \cdot 37 \cdot 79 \cdot 97 \cdot 103 \cdot 331 \cdot 601 \cdot 619 \cdot 769 \cdot 907 \cdot 1\,063$, respectively. The discriminant of f_1 equals $2^{12} \cdot 3^2 \cdot 5^2 \cdot 13 \cdot 17 \cdot 17\,722\,398\,737 \cdot c273$, for a 273-digit composite integer $c273$ that is most likely free of squares and of factors less than 10^{40}. The discriminant of f_2 equals one. A renewed search at EPFL in the spring of 2007 (also not using the idea from [16]) produced a couple of candidates of similar quality, again after spending about 20 core years.

Following [15], during the search, the leading coefficient of f_2 allowed 11 (search at BSI) or 10 (search at EPFL) prime factors equal to $1 \bmod 6$ and at most one other factor $< 2^{15.5}$. The leading coefficient of f_1 was a multiple of $258\,060 = 2^2 \cdot 3 \cdot 5 \cdot 11 \cdot 17 \cdot 23$. At least $2 \cdot 10^{18}$ pairs (f_1, f_2) were considered.

2.3 Sieving

To be able to profit from near misses during the search for relations an integer x is defined to be (b_k, b_ℓ)-smooth if with the exception of, say, four prime factors between b_k and b_ℓ, all remaining prime factors of $|x|$ are at most b_k. We thus change the definition of a relation into a coprime pair of integers (a, b) with $b > 0$ such that $b^{d_1} f_1(a/b)$ is (b_1, b_ℓ)-smooth and $b^{d_2} f_2(a/b)$ is (b_2, b_ℓ)-smooth. Although *large primes* speed up the sieving, they make it harder to decide whether enough relations have been found, as the criterion that more than $\pi(b_1)+\pi(b_2)+2$ are needed is no longer adequate. The decision requires duplicate and singleton removal. It is briefly touched upon at the end of Section 2.3.

We used $b_1 = 11 \cdot 10^8$, $b_2 = 2 \cdot 10^8$ and $b_\ell = 2^{40}$ on cores with at least 2 GB RAM (the majority) and $b_1 = 4.5 \cdot 10^8$, $b_2 = 10^8$ on others (preferably with at least a GB RAM). Based on sieving experiments it was expected that it would suffice to use as sieving region the subset S of $\mathbf{Z} \times \mathbf{Z}_{>0}$ of about $11 \cdot 10^{18}$ coprime pairs (a, b) with $|a| \leq 3 \cdot 10^9 \cdot \kappa^{1/2} \approx 6.3 \cdot 10^{11}$ and $0 < b \leq 3 \cdot 10^9 / \kappa^{1/2} \approx 1.4 \cdot 10^7$. Here $\kappa = 44\,000$ approximates the *skewness* of f_1. It is used to approximately minimize the largest norm $v_1(a, b)$ encountered in the sieving region. Although prime ideal norms up to 2^{40} were accepted, the parameters were optimized for norms up to 2^{37}. Most jobs attempted to factor after the sieving algebraic and rational cofactors up to 2^{140} and 2^{110}, respectively, only considering the most promising candidates [14]. As far as we know, this was the first NFS factorization allowing more than three algebraic large primes.

Disregarding factors of f_k's discriminant, a prime p dividing $f_k(r)$ is equivalent to $(r \bmod p)$ being a root of f_k modulo p. Because $d_2 = 1$, the polynomial f_2 has one root modulo p for each prime p not dividing its leading coefficient, and each such p divides $f_2(j)$ once every p consecutive j-values. For f_1 there may be between zero to d_1 roots modulo p: some primes p do not divide $f_1(j)$ for any j, whereas other p may divide $f_1(j)$ a total of d_1 times for every p consecutive j-values. The (p, r) pairs with $p \leq b_1$ for f_1 and $p \leq b_2$ for f_2 are precomputed.

Early implementations of NFS used *line sieving*: for some b-value and k, one marks for each precomputed (p, r) pair for f_k the a-values of the form $rb + ip$ for $i \in \mathbf{Z}$ with "p," since for those a-values p divides $b^{d_k} f(a/b) = v_k(a, b)$. The locations hit by many different p's are remembered, and the process is repeated

for the other k. Relations may be found at locations that were hit twice. With many lines (b-values) to be processed, line sieving can easily be parallelized.

For RSA-768 we did not use line sieving but a more efficient approach that has gained popularity since the mid 1990s: the *lattice sieve* as described in [30]. For a (prime,root) pair $\mathfrak{q} = (q, s)$ define $L_\mathfrak{q}$ as the lattice of integer linear combinations of the 2-dimensional integer (row-)vectors $(q, 0), (s, 1) \in \mathbf{Z}^2$. Let $S_\mathfrak{q} = S \cap L_\mathfrak{q}$. Fix a (prime,root) pair $\mathfrak{q} = (q, s)$ for, say, f_1. The *special prime* q (as it was referred to in [30]) is chosen smaller than b_ℓ, and it divides $b^{d_1} f_1(a/b)$ for $(a, b) \in S_\mathfrak{q}$. Lattice sieving consists of marking, for each precomputed (prime,root) pair $\mathfrak{p}$ for f_1, the points in the intersection $L_\mathfrak{p} \cap S_\mathfrak{q}$. Locations that are hit often are remembered, and the process is repeated for the precomputed (prime,root) pairs for f_2. Relations may be found at locations that were hit twice. For each relation thus found, q divides $v_1(a, b)$. The process is repeated for other $\mathfrak{q}$ until enough relations have been found. Because relations may be found for each special prime occurring in $v_1(a, b)$, duplicates will be found when lattice sieving.

In practice one fixes bounds I and J independent of $\mathfrak{q}$ and defines $S_\mathfrak{q} = \{iu + jv : i, j \in \mathbf{Z}, -I/2 \le i < I/2, 0 < j < J\}$, where u, v form a basis for $L_\mathfrak{q}$ that minimizes the norms $v_1(a, b)$ for $(a, b) \in S_\mathfrak{q}$. Such a basis is found by partially reducing the basis $(q, 0), (s, 1)$ for $L_\mathfrak{q}$ such that the first coordinate is roughly κ times bigger than the second, cf. skewness of S. Sieving is carried out over the set $\{(i, j) \in \mathbf{Z} \times \mathbf{Z}_{>0} : -I/2 \le i < I/2, 0 < j < J\}$, interpreted as $S_\mathfrak{q}$.

We used $I = 2^{16}$ and $J = 2^{15}$, i.e., a lattice sieving area of size roughly $2^{31} \approx 2 \cdot 10^9$. With $b_1 = 11 \cdot 10^8$ and $b_2 = 2 \cdot 10^8$, the majority of the sieving-primes can be expected to hit $S_\mathfrak{q}$ only a few times. Thus, for any sieving-$\mathfrak{p}$, only a few of the j-values (the lines) will be hit, unlike line sieving where each line will be hit several times by each prime. Therefore, when lattice sieving, a more sophisticated sieving method must be used that avoids looking at all lines $0 < j < J$ for each $\mathfrak{p}$. This *sieving by vectors* [30] was first implemented in [13] and used for many factorizations in the 1990s [10,7]. We used the implementation from [12], described in Appendix A. Most of the about 0.48 billion (prime,root) pairs (q, s) for special primes q between 0.45 and 11.1 billion (and some special primes below 0.45 billion, with a smaller b_1-value) were processed by eight contributing parties (cf. Table 1) during the period August 2007 until April 2009. Scaled to a 2.2 GHz Opteron core with 2 GB RAM, a single $L_\mathfrak{q}$ was processed in less than 100 seconds on average and produced about 134 relations, for an average of about four relations every three seconds. This average rate varies by a factor of about two between both ends of the special primes range that we used.

We collected 64 334 489 730 relations in total, each requiring about 150 bytes. Compressed they occupied about 5 terabytes of disk space, backed up at various locations. The 27.4% duplicates were removed using hashing. This was done mostly during the sieving, overall taking less than 10 days on a 2.66 GHz Core2 processor with ten 1TB hard disks. After including 57 223 462 *free relations* [19], we ended up with 47 762 243 404 relations involving 35 288 334 017 prime ideals.

Given the set of unique relations, those that involve a prime ideal that does not occur in any other relation, the singletons, cannot be part of a dependency.

Table 1: Percentages contributed.

contributor	relations contribution	matrix stages, % of matrix effort			
		1 (60%)	2 (0%)	3 (40%)	total
Bonn (University and BSI)	8.14%				
CWI	3.44%				
EPFL	29.78%	34.3%	100%	78.2%	51.9%
INRIA LORIA (ALADDIN-G5K)	37.97%	46.8%		17.3%	35.0%
NTT	15.01%	18.9%		4.5%	13.1%
Scott Contini (Australia)	0.43%				
Paul Leyland (UK)	0.69%				
Heinz Stockinger (Enabling Grids for E-sciencE)	4.54%				

Singletons were removed using hashing. Doing this once reduced the set of relations to 28 984 986 047 elements with 14 498 007 183 prime ideals. Removal of singletons usually creates new singletons, and the process must be repeated until no new singletons are created. After a few more singleton removals 24 615 168 385 relations involving at most 9 976 671 468 prime ideals were left.

Further singleton removal was combined with *clique removal* [6], i.e., search of combinations with matching first degree prime ideals of norms larger than b_k. Ultimately, this led to 2 458 248 361 relations with 1 697 618 199 prime ideals, still containing an almost insignificant number (604 423) of free relations. Since there are more relations than prime ideals (so that dependencies exist), we had done enough sieving and lots of flexibility to create a matrix. Singleton and clique removal took less than 10 days on the same platform as above.

2.4 The matrix step

Current best methods to find dependencies among the rows of a sparse matrix take time proportional to the product of the dimension and the weight (i.e., number of non-zero entries) of the matrix. Merging is a generic term for the set of strategies developed to build a matrix for which close to optimal dependency search can be expected. It is described in [6]. We ran about 10 separate merging jobs, aiming for various optimizations (low dimension, low weight, best-of-both, etc.), which each took a couple of days on a single core per node of a 37-node 2.66 GHz Core2 cluster with 16 GB RAM per node, and a not particularly fast interconnection network. The best alternative was a $192\,796\,550 \times 192\,795\,550$-matrix of total weight 27 797 115 920 (on average 144 non-zeros per row), requiring about 105 GB. It was generated in 5-days on two to three cores on the 37-node cluster, where the long duration was probably due to the large communication overhead. When we started the project, we expected dimension about a quarter billion and density per row of about 150, which would have been about $\frac{7}{4}$ times harder.

To find dependencies we used block Wiedemann [9,38] as described in [2, Section 5.1] . We give a high level description [18, Section 2.19]. Given a non-singular $d \times d$ matrix M over the finite field $\mathbf{F}_2$ and $b \in \mathbf{F}_2^d$, we wish to solve the system $Mx = b$. The minimal polynomial F of M on the vector space spanned by b, Mb, M^2b, ... has degree at most d, so that $F(M)b = \sum_{i=0}^{d} F_i M^i b = 0$. From $F_0 = 1$ it follows that $x = \sum_{i=1}^{d} F_i M^{i-1} b$, so it suffices to find the F_i's.

Denoting by $m_{i,j}$ the jth coordinate of the vector $M^i b$, it follows that for each j with $1 \le j \le d$ the sequence $(m_{i,j})_{i=0}^{\infty}$ satisfies a linear recurrence relation of order at most d defined by the coefficients F_i: for any $t \ge 0$ and $1 \le j \le d$ we have that $\sum_{i=0}^{d} F_i m_{i+t,j} = 0$. Given $2d+1$ consecutive terms of an order d linear recurrence, its coefficients can be computed using the Berlekamp-Massey method [22,38]. Each j may lead to a polynomial of smaller degree than F, but taking, if necessary, the least common multiple of the polynomials found for a few different indices j, the correct minimal polynomial will be found.

Summarizing the above, there are three major stages: a first iteration consisting of $2d$ matrix×vector steps to generate $2d+1$ terms of the linear recurrence, the Berlekamp-Massey stage to calculate the F_i's, and a second iteration consisting of d matrix×vector steps to calculate the solution using the F_i's. For large matrices the first and the final stage are the most time consuming.

In practice it is common to use *blocking*, to take advantage of the fact that on 64-bit machines 64 different vectors b over $\mathbf{F}_2$ can be processed simultaneously, at little or no extra cost compared to a single vector [9], while using the same three main stages. If the vector $\bar{b}$ is 64 bits wide and in the first stage the first 64 coordinates of each of the generated 64 bits wide vectors $M^i\bar{b}$ are kept, the number of matrix (M) times vector ($\bar{b}$) multiplications in both the first and the final stage is reduced by a factor of 64 compared to the number of M times b multiplications, while making the central Berlekamp-Massey stage a bit more cumbersome. It is less common to take the blocking a step further and run both iteration stages spread over a small number n' of different sequences, possibly run on disjoint clusters; in [2] this was done with $n' = 4$ sequences run on three clusters. If for each sequence one keeps the first $64 \cdot n'$ coordinates of each of the 64 bits wide vectors they generate during the first stage, the number of steps to be carried out (per sequence) is further reduced by a factor of n', while allowing independent and simultaneous execution on possibly n' disjoint clusters. After the first stage the data generated for the n' sequences have to be gathered at a central location where the Berlekamp-Massey stage will be carried out.

While keeping the first $64 \cdot n'$ coordinates per step for each sequence results in a reduction of the number of steps per sequence by a factor of $64 \cdot n'$, keeping a different number of coordinates while using n' sequences results in another reduction in the number of steps for the first stage. Following [2, Section 5.1], if the first $64 \cdot m'$ coordinates are kept of the 64 bits wide vectors for n' sequences, the numbers of steps become $\frac{d}{64 \cdot m'} + \frac{d}{64 \cdot n'} = (\frac{n'}{m'} + 1)\frac{d}{64 \cdot n'}$ and $\frac{d}{64 \cdot n'}$ for the first and third stage, respectively and for each of the n' sequences. The choices of m' and n' should be weighed off against the cost of the Berlekamp-Massey step with time and space complexities proportional to $\frac{(m'+n')^3}{n'} d^{1+o(1)}$ and $\frac{(m'+n')^2}{n'} d$, respectively and for $d \to \infty$, and where the exponent "3" may be replaced by the matrix multiplication exponent (our implementation uses "3").

When running the first stage using n' sequences, the effect of non-identical resources used for different sequences quickly becomes apparent: some locations finish their work faster than others (depicted in Fig. 1). To keep the fast contributors busy and to reduce the work of the slower ones (thereby reducing the

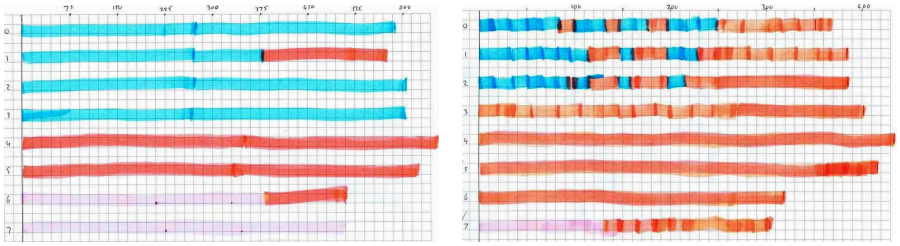
wall-clock time), a quickly processed first stage sequence may continue for s steps beyond $(\frac{n'}{m'}+1)\frac{d}{64\cdot n'}$ while reducing the number of steps in another first stage sequence by the same s. As described in Appendix B, this can be done in a very flexible way, as long as the overall number of steps over all first stage sequences adds up to $n' \cdot (\frac{n'}{m'}+1)\frac{d}{64\cdot n'}$. The termination points of the sequences in the third stage need to be adapted accordingly. This is easily arranged for, since the third stage allows much easier and much wider parallelization anyhow (assuming checkpoints from the first stage are kept). Another way to keep all parties busy is swapping jobs, thus requiring data exchanges, synchronization, and more human interaction, making it a less attractive option altogether.

For our matrix with $d \approx 193 \cdot 10^6$ we used, as in [2], $m' = 2n'$. But where $n' = 4$ was used in [2], we used $n' = 8$. This quadrupled the Berlekamp-Massey runtime and doubled its memory compared to the matrix from [2], on top of the increased runtime and memory demands caused by the larger dimension of the matrix. On the other hand, the compute intensive first and third stages could be split up into twice as many independent jobs as before. For the first stage on average $(\frac{8}{16}+1)\frac{193\cdot 10^6}{64\cdot 8} \approx 565\,000$ steps needed to be taken per sequence (for 8 sequences), for the third stage the average was about $\frac{193\cdot 10^6}{64\cdot 8} \approx 380\,000$ steps. The actual numbers of steps varied, approximately, between 490 000 and 650 000 for the first stage and between 300 000 and 430 000 for the third stage. The calculation of these stages was carried out on a wide variety of clusters accessed from three locations: a 56-node cluster of 2.2GHz dual hex-core AMD processors with Infiniband at EPFL (installed while the first stage was in progress), a variety of ALADDIN-G5K clusters in France accessed from INRIA LORIA, and a cluster of 110 3.0GHz Pentium-D processors on a Gb Ethernet at NTT.

On 12 nodes of a 12-cores-per-node cluster of 2.2 GHz AMD processors with 16 GB RAM per node and an Infiniband network, one multiplication step (of the matrix times a 64 bits wide vector) took between 4.3 and 4.5 seconds for the first stage and about 4.8 seconds for the slightly more involved third stage. Per-iteration timings for stage one on the Pentium cluster are 11.6 seconds per iteration when two sequences are run in parallel (thus, effectively, 5.8 seconds per sequence), and 6.4 seconds if one sequence is processed. For the third stage it was 7.8 seconds per iteration, for a single sequence. For the ALADDIN-G5K clusters the per-iteration timings for stages one and three varied between 2.3 and 4.1 seconds, and between 2.6 and 17.9 seconds, respectively. It follows that doing the entire first and third stage would have taken 98 days on 48 nodes (576 cores) of the 56-node EPFL cluster.

The first stage was split up into eight independent jobs run in parallel on those clusters, check-pointing once every 2^{14} steps. Running a first (or third) stage sequence required 180 GB RAM, a single 64 bits wide $\bar{b}$ took 1.5 GB, and a single m_i matrix 8 KB, of which 565 000 were kept, on average, per first stage sequence. Each partial sum during the third stage evaluation required 12 GB.

The central Berlekamp-Massey stage was done in 17 hours and 20 minutes on the 56-node EPFL cluster (with 16 GB RAM per node), while using just 4 of the 12 available cores per node. Most of the time the available 896 GB RAM sufficed,



(a) First stage contributions. (b) Final shot of third stage bookkeeping.

Fig. 1: Contributions to sequences 0-7: blue=INRIA, orange=EPFL, pink=NTT.

but during a central part of the calculation more memory was needed (up to about 1 TB) and some swapping occurred. The third stage started right after completion of the second stage, running as many jobs in parallel as possible. The actual bookkeeping sheet used is pictured in Fig. 1b. Fig. 1a pictures the first stage contributions apocryphally but accurately. Calendar time for the entire block Wiedemann step was 119 days, finishing on December 8, 2009.

2.5 That's a bingo[7]

As expected the matrix step resulted in $512 = 64 \cdot 8$ linear dependencies modulo 2 among the exponent vectors, more than enough to include the quadratic characters at this stage [1]. This reduced the solution space to 460 elements, giving us that many independent chances of about $\frac{1}{2}$ to factor RSA-768. In the $52 = 512 - 460$ difference, a dimension of 46 can be attributed to prime ideals not included in the matrix that divide the leading coefficients or the discriminant.

The square roots of the algebraic numbers were calculated by means of the method from [23] (see also [29]), which uses the known factorization of the algebraic numbers into small prime ideals of known norms. The implementation based on [3] turned out to have a bug when computing the valuations for the free relations of the prime ideals lying above the divisor $17\,722\,398\,737 > 2^{32}$ of the discriminant of f_1. Along with a bug in the quadratic character calculation, this delayed completion of the square root step by a few (harrowing) days.

Once the bugs were located and fixed, it took two hours using the hard disk and one core on each of twelve dual hex-core 2.2GHz AMD processors to compute the exponents of all prime ideals for eight solutions simultaneously. Computing a square root using the implementation from [3] took one hour and forty minutes on such a dual hex-core processor. The first one (and four of the other seven) led to the factorization $p \cdot q$, found at 20:16 GMT on December 12, 2009:

[7] "Is that the way you say it? "That's a bingo?""
"You just say "bingo"." [37]

$$p = 33478071698956898786044169848212690817704794983713768568912431388982883793878002287614711652531743087737814467999489,$$
$$q = 36746043666799590428244633799627952632279158164343087642676032283815739666511279233373417143396810270092798736308917,$$

where p and q are 384-bit, 116-digit primes. With "pk" a k-digit prime, we found:

$$p - 1 = 2^8 \cdot 11^2 \cdot 13 \cdot 7193 \cdot 160378082551 \cdot 7721565388263419219 \cdot 11110316344948488248471139305 3 \cdot p47,$$
$$p + 1 = 2 \cdot 3 \cdot 5 \cdot 319321227495533722620054918616303451834164 67 \cdot p71,$$
$$q - 1 = 2^2 \cdot 359 \cdot p113, \quad q + 1 = 2 \cdot 3 \cdot 23 \cdot 41 \cdot 47 \cdot 239875144072757917901 \cdot p90.$$

3 Concluding remarks

It is customary to conclude a paper reporting a new factoring record with a preview of coming attractions. Our main conclusion was summarized in the introduction and was already announced in [2, Section 7]: at this point factoring a 1024-bit RSA modulus looks more than five times easier than a 768-bit RSA modulus looked back in 1999, when we achieved the first public factorization of a 512-bit RSA modulus. Nevertheless, a 1024-bit RSA modulus is still about a thousand times harder to factor than a 768-bit one. It may be possible to factor a 1024-bit RSA modulus within the next decade by means of an academic effort on the same scale as the effort presented here. Recent standards recommend phasing out such moduli by the end of the year 2010 [28]. See also [21].

Another conclusion from our work is that we can confidently say that if we restrict ourselves to an open community, academic effort such as ours and unless something dramatic happens in factoring, we will not be able to factor a 1024-bit RSA modulus within the next five years [27]. After that, all bets are off.

The ratio between sieving and matrix time was almost 10. This is probably not optimal if one wants to minimize the overall runtime. But the latter may not be the most important criterion. Sieving is easy, and doing more of it may be a good investment if that leads to an easier matrix step. The relations collected for RSA-768 will give us a better insight in the trade-off between sieving and matrix efforts, where also the choice of the large prime bound b_ℓ may play a role. This is a subject for further study that may be expected to lead, ultimately, to a recommendation for close to optimal parameter choices – depending on what one wants to optimize – for NFS factorizations in the 700- to 800-bit range.

Our computation required more than 10^{20} operations. With the equivalent of almost 2000 years of computing on a single core 2.2GHz AMD Opteron, on the order of 2^{67} instructions were carried out. The overall effort is sufficiently low that even for short-term protection of data of little value, 768-bit RSA moduli can no longer be recommended. This conclusion is the opposite of the one on [36], which is based on a hypothetical factoring effort of six months on 100 000 workstations, i.e., about two orders of magnitude more than we spent.

Acknowledgements. This work was supported by the Swiss National Science Foundation under grant numbers 200021-119776 and 206021-128727 and by the Netherlands Organization for Scientific Research (NWO) as project 617.023.613. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies (see `https://www.grid5000.fr`). Condor middleware was used on EPFL's Greedy network. We acknowledge the help of Cyril Bouvier during filtering and merging experiments. We gratefully acknowledge sieving contributions by BSI, Scott Contini (using resources provided by AC3, the Australian Centre for Advanced Computing and Communications), Paul Leyland (using teaching lab machines at the Genetics Department of Cambridge University), and Heinz Stockinger (using EGEE, Enabling Grids for E-sciencE). Parts of this paper were inspired by Col. Hans Landa.

References

1. L. M. Adleman. Factoring numbers using singular integers. In *STOC*, pages 64–71. ACM, 1991.
2. K. Aoki, J. Franke, T. Kleinjung, A. K. Lenstra, and D. A. Osvik. A kilobit special number field sieve factorization. In *Asiacrypt 2007*, volume 4833 of *LNCS*, pages 1–12, 2007.
3. F. Bahr. Liniensieben und Quadratwurzelberechnung für das Zahlkörpersieb, 2005. Diplomarbeit, University of Bonn.
4. F. Bahr, M. Böhm, J. Franke, and T. Kleinjung. Factorization of RSA-200. `http://www.loria.fr/~zimmerma/records/rsa200`, May 2005.
5. J. Buhler, P. Montgomery, R. Robson, and R. Ruby. Implementing the number field sieve. Technical report, Oregon State University, 1994.
6. S. Cavallar. Strategies in filtering in the number field sieve. In *ANTS IV*, volume 1838 of *LNCS*, pages 209–232, 2000.
7. S. Cavallar, B. Dodson, A. K. Lenstra, W. M. Lioen, P. L. Montgomery, B. Murphy, H. J. J. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. C. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, and P. Zimmermann. Factorization of a 512-bit RSA modulus. In *Eurocrypt 2000*, volume 1807 of *LNCS*, pages 1–18, 2000.
8. D. Coppersmith. Solving linear equations over GF(2): block Lanczos algorithm. *Linear Algebra and its Applications*, 192:33–60, 1993.
9. D. Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math. Comput.*, 62(205):333–350, 1994.
10. J. Cowie, B. Dodson, R. M. Elkenbracht-Huizing, A. K. Lenstra, P. L. Montgomery, and J. Zayer. A world wide number field sieve factoring record: On to 512 bits. In *Asiacrypt 1996*, volume 1163 of *LNCS*, pages 382–394, 1996.
11. J. D. Dixon. Asymptotically fast factorization of integers. *Math. Comp.*, 36:255–260, 1981.
12. J. Franke and T. Kleinjung. Continued fractions and lattice sieving. In *Workshop record of SHARCS 2005*, 2005. `http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/talks/FrankeKleinjung.pdf`.
13. R. A. Golliver, A. K. Lenstra, and K. S. McCurley. Lattice sieving and trial division. In *ANTS I*, volume 877 of *LNCS*, pages 18–27, 1994.

14. T. Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers. In *Workshop record of SHARCS*, 2005. `http://www.hyperelliptic.org/tanja/SHARCS/talks06/thorsten.pdf`.
15. T. Kleinjung. On polynomial selection for the general number field sieve. *Math. Comp.*, 75:2037–2047, 2006.
16. T. Kleinjung. Polynomial selection. presented at the CADO workshop on integer factorization,`http://cado.gforge.inria.fr/workshop/slides/kleinjung.pdf`, 2008.
17. A. K. Lenstra. Computational methods in public key cryptology. In *Coding theory and cryptology*, Lecture Notes Series, pages 175–238. Institute for Mathematical Sciences, National University of Singapore, 2002.
18. A. K. Lenstra and H. W. Lenstra Jr. Algorithms in number theory. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 673–716. Elsevier, 1990.
19. A. K. Lenstra and H. W. Lenstra, Jr. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verslag, 1993.
20. A. K. Lenstra, H. W. Lenstra Jr., M. S. Manasse, and J. M. Pollard. The factorization of the ninth Fermat number. *Math. of Comp.*, 61(203):319–349, 1993.
21. A. K. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, and P. C. Leyland. Factoring estimates for a 1024-bit RSA modulus. In *Asiacrypt 2003*, volume 2894 of *LNCS*, pages 55–74, 2003.
22. J. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans Information Theory*, 15:122–127, 1969.
23. P. Montgomery. Square roots of products of algebraic numbers. `http://ftp.cwi.nl/pub/pmontgom/sqrt.ps.gz`.
24. P. Montgomery and B. Murphy. Improved polynomial selection for the number field sieve. Technical report, the Fields institute, Toronto, Ontario, Canada., June 1999.
25. M. A. Morrison and J. Brillhart. A method of factoring and the factorization of F_7. *Math. of Comp.*, 29(129):183–205, 1975.
26. B. Murphy. Modelling the yield of number field sieve polynominals. In *ANTS III*, volume 1423 of *LNCS*, pages 137–150, 1998.
27. National Institute of Standards and Technology. Discussion paper: the transitioning of cryptographic algorithms and key sizes. `http://csrc.nist.gov/groups/ST/key_mgmt/documents/Transitioning_CryptoAlgos_070209.pdf`.
28. National Institute of Standards and Technology. Special publication 800-57: Recommendation for key management part 1: General (revised). `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf`.
29. P. Q. Nguyen. A Montgomery-like square root for the number field sieve. In *ANTS III*, volume 1423 of *LNCS*, pages 151–168, 1998.
30. J. M. Pollard. The lattice sieve. pages 43–49 in [19].
31. C. Pomerance. Analysis and comparison of some integer factoring algorithms. In H. W. Lenstra, Jr. and R. Tijdeman, editors, *Computational Methods in Number Theory*, pages 89–139. Math. Centrum Tract 154, Amsterdam, 1982.
32. C. Pomerance. The quadratic sieve factoring algorithm. In *Eurocrypt 1984*, volume 209 of *LNCS*, pages 169–182, 1984.
33. C. Pomerance. A tale of two sieves. `http://www.ams.org/notices/199612/pomerance.pdf`, 1996.
34. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

35. RSA the security division of EMC. The RSA challenge numbers. formerly on `http://www.rsa.com/rsalabs/node.asp?id=2093`, now on `http://en.wikipedia.org/wiki/RSA_numbers`.
36. RSA the security division of EMC. The RSA factoring challenge FAQ. `http://www.rsa.com/rsalabs/node.asp?id=2094`.
37. Q. Tarantino. That's a bingo! `http://www.youtube.com/watch?v=WtHTc8wIo4Q`, `http://www.imdb.com/title/tt0361748/quotes`.
38. E. Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *Journal of symbolic computation*, 33(5):757–775, 2002.

A Sieving by vectors

We briefly describe the lattice sieve implementation from [12] which was used for most NFS factorization records of the last decade.

Let $v_k(a,b) = b^{d_k} f_k(a/b)$. Lattice sieving, introduced by Pollard [30], increases the smoothness probability of $v_1(a,b)$ by looking at (a,b)-pairs for which $v_1(a,b)$ is divisible by a large special prime q. Let $s \bmod q$ be a residue class such that this is the case for $a \equiv sb \bmod q$. One constructs a reduced basis (u,v) of the lattice of all $(a,b) \in \mathbf{Z}^2$ with $a \equiv sb \bmod q$. A scalar product adapted to the skewness of the polynomial pair is used for this reduction. The problem is then to find all coprime pairs (i,j), $-I/2 \le i < I/2$, $0 < j < J$, such that $v_1(a,b)/q$ and $v_2(a,b)$ are smooth, with $(a,b) = iu + jv$. We assume I to be even. As mentioned in Section 2.3, for practical values of the parameters, I is much smaller than the smoothness bounds b_1 and b_2, and it is non-trivial to efficiently sieve such regions.

Pollard proposed to do this by using, for each (prime,root) pair $\mathfrak{p}$ with prime p bounded by the relevant smoothness bound b_k, a reduced base of the lattice $\Gamma_{\mathfrak{p}}$ of pairs (i,j) for which $v_k(a,b)$ for the corresponding (a,b)-pair is divisible by p. In [13] that approach was used for p larger than a small multiple of I, while avoiding storage of "even, even" sieve locations (and using line sieving for the other primes). Our approach uses a truncated continued fraction expansion to determine a basis $B = \big((\alpha,\beta),(\gamma,\delta)\big)$ of $\Gamma_{\mathfrak{p}}$ with the following properties:

a The numbers β and δ are positive.
b We have $-I < \alpha \le 0 \le \gamma < I$ and $\gamma - \alpha \ge I$.

Let us assume that $\Gamma_{\mathfrak{p}}$ consists of all (i,j) for which $i \equiv \rho j \bmod p$, where $0 < \rho < p$. The case $\rho = 0$ and the case where $\Gamma_{\mathfrak{p}}$ consists of all (i,j) for which p divides j are not treated, because they produce just $(0,1)$ and $(1,0)$, respectively, as only coprime pairs. We also assume $p \ge I$, as smaller primes are better treated by line sieving. To construct a basis with the above properties, one takes $(i_0,j_0) = (-p,0)$, $(i_1,j_1) = (\rho,1)$ and puts $(i_{\ell+1},j_{\ell+1}) = (i_{\ell-1},j_{\ell-1}) + r(i_\ell,j_\ell)$ with $r = \lfloor -\frac{i_{\ell-1}}{i_\ell} \rfloor$. Note that $(-1)^{\ell+1} i_\ell \ge 0$, that r is positive and that the j_ℓ thus form an increasing sequence of non-negative numbers. The process is stopped at the first ℓ with $\mid i_\ell \mid < I$. If ℓ is odd, we put $(\alpha,\beta) = (i_{\ell-1},j_{\ell-1}) + r(i_\ell,j_\ell)$, where r is the smallest integer for which $\alpha > -I$. If ℓ is even, we put $(\gamma,\delta) =$

$(i_{\ell-1}, j_{\ell-1}) + r(i_\ell, j_\ell)$, where r is the smallest integer such that $\gamma < I$. In both cases, the element of $B = \big((\alpha, \beta), (\gamma, \delta)\big)$ not yet described is given by (i_ℓ, j_ℓ).

To explain how to efficiently sieve using a basis with these properties, let $(i, j) \in \mathit{\Gamma}_{\mathfrak{p}}$ such that $-I/2 \le i < I/2$. We want to find the (uniquely determined) $(i', j') \in \mathit{\Gamma}_{\mathfrak{p}}$ such that $-I/2 \le i' < I/2$, $j' > j$, and j' is as small as possible. As B is a basis of $\mathit{\Gamma}_{\mathfrak{p}}$, there are integers d and e with

$$(i', j') - (i, j) = d(\alpha, \beta) + e(\gamma, \delta).$$

If $d \cdot e < 0$, then condition **b** on B would force the first component of the right hand side to have absolute value $\ge I$, whereas our constraints on i and i' force it to have absolute value $< I$. Since $j' - j$, β, and δ are all positive, we have $d \ge 0$ and $e \ge 0$. It is now easy to see that the solution to our problem is:

$$(d, e) = \begin{cases} (0, 1) & \text{if } i < I/2 - \gamma \\ (1, 1) & \text{if } I/2 - \gamma \le i < -I/2 - \alpha \\ (1, 0) & \text{if } i \ge -I/2 - \alpha. \end{cases}$$

The minimality of j' follows because $d = 0$ leads to a violation of $i' < I/2$ unless $i < I/2 - \gamma$ (i.e., save for the first of the above cases) and $e = 0$ leads to $i' < -I/2$ unless $i \ge -I/2 - \alpha$ (i.e., save for the third of the above cases).

To implement this process on a modern CPU, it seems best to take $I = 2^\iota$ for some natural number ι. It is possible to identify pairs (i, j) of integers with $-I/2 \le i < I/2$ with integers x by putting $x = j \cdot I + i + I/2$. If $x' = j' \cdot I + i' + I/2$ with (i', j') as above, then $x' = x + C$, $x' = x + A + C$ and $x' = x + A$ in the three cases above, with $A = \alpha + I \cdot \beta$ and $C = \gamma + I \cdot \delta$. The first component of a pair (i, j), (α, β) or (γ, δ) is extracted from these numbers by using a bitwise logical operation, and the selection of the appropriate one of the above three cases is best done using conditional move instructions.

For cache efficiency, the sieving region $S_{\mathfrak{q}}$ was split into areas A_t, $0 \le t < T$, of size equal to the L1-cache size. For primes p larger than that size (or a small multiple thereof), sieving is not done directly. Instead, the numbers x corresponding to elements of $S_{\mathfrak{q}} \cap \mathit{\Gamma}_{\mathfrak{p}}$ were calculated ahead of the sieving process, and their offsets into the appropriate region A_t stored in the corresponding element of an array $\mathcal{S}$ of T stacks. To implement the trial division sieve efficiently, the corresponding factor base index was also stored. Of course, this approach may also be used for line sieving, and in fact was used in [3]. A similar approach has been described by T. Oliveira e Silva in connection with his implementation of the Odlyzko-Lagarias-Lehmer-Meissel method.

Parallelization is possible in several different ways. A topology for splitting the sieving region among several nodes connected by a network is described in [12]. If one wants to split the task among several cores sharing their main memory, it seems best to distribute the regions A_t and also the large factor base primes among them. Each core first calculates its part of $\mathcal{S}$, for its assigned part of the large factor base elements, and then uses the information generated by all cores to treat its share of regions A_t. A lattice siever parallelized that way was used for a small part of the RSA-576 sieving tasks, but the code fell out of

use and was not used for the current project. The approach may be more useful today, with many cores per processor being a standard.

B Unbalanced sequences in block Wiedemann

Before describing the modification for unbalanced sequence lengths we give an overview of Coppersmith's block version of the Berlekamp-Massey algorithm. To avoid a too technical description we simplify the presentation of Coppersmith's algorithm and refer to [9] for details. The modification was also be applied to Thomé's subquadratic algorithm [38]. Below m and n are as in [9], and the terms "$+O(1)$" are constants depending on m and n. We assume that m and n, which play the role of $64 \cdot m'$ and $64 \cdot n'$ in Section 2.4, are much smaller than d.

Let M be a $d \times d$ matrix over $\mathbf{F}_2$, $m \geq n$, $x_k \in \mathbf{F}_2^d$, $1 \leq k \leq m$ and $y_j \in \mathbf{F}_2^d$, $1 \leq j \leq n$ satisfying certain conditions. Set $a_{j,k}^{(i)} = x_k^T M^i y_j$ and

$$A = \sum_i (a_{j,k}^{(i)}) X^i \quad \in \mathrm{Mat}_{n,m}[X].$$

In the first step we calculate the coefficients of A up to degree $\frac{d}{m} + \frac{d}{n} + O(1)$.

The goal of the Berlekamp-Massey step is to find polynomials of matrices $F \in \mathrm{Mat}_{n,n}[X], G \in \mathrm{Mat}_{n,m}[X]$ with $\deg(F) \leq \frac{d}{n} + O(1)$, $\deg(G) \leq \frac{d}{n} + O(1)$ and

$$FA \equiv G \pmod{X^{\frac{d}{m} + \frac{d}{n} + O(1)}}.$$

Intuitively, we want to produce at least d zero rows in the higher coefficients of FA up to degree $\frac{d}{m} + \frac{d}{n} + O(1)$. Writing $F = \sum_{i=0}^{d_F} (f_{j,k}^{(i)}) X^i$, $d_F = \deg(F)$ the jth row of coefficient $d_F + b$ of G being zero corresponds to

$$(M^b x_h)^T v_j = 0 \quad \text{for } 1 \leq h \leq m, 0 \leq b < \frac{d}{m} + O(1) \text{ where}$$

$$v_j = \sum_{k=1}^{n} \sum_{i=0}^{d_F} f_{j,k}^{(d_F - i)} \cdot M^i y_k.$$

Coppersmith's algorithm produces a sequence of matrices (of $m + n$ rows) $F_t \in \mathrm{Mat}_{m+n,n}[X]$ and $G_t \in \mathrm{Mat}_{m+n,m}[X]$ for $t = t_0, \ldots, \frac{d}{m} + \frac{d}{n} + O(1)$ (where $t_0 = O(1)$) such that

$$F_t A \equiv G_t \pmod{X^t}$$

and the degrees of F_t and G_t are roughly $\frac{m}{m+n} t$. In a first step t_0 and F_{t_0} are chosen such that certain conditions are satisfied, in particular that $\deg(F_{t_0}) = O(1)$ and $\deg(G_{t_0}) = O(1)$. To go from t to $t + 1$ a polynomial of degree 1 of matrices $P_t \in \mathrm{Mat}_{m+n,m+n}[X]$ is constructed and we set $F_{t+1} = P_t F_t$ and $G_{t+1} = P_t G_t$. This construction is done as follows. We have $F_t A \equiv G_t + E_t X^t \pmod{X^{t+1}}$ for some matrix E_t. Respecting a restriction involving the degrees of the rows of G_t (essentially we avoid destroying previously constructed zero

rows in the G's) we perform a Gaussian elimination on E_t, i.e., we obtain $\tilde{P}_t$ such that

$$\tilde{P}_t E_t = \begin{pmatrix} 0 \\ \mathbf{1}_m \end{pmatrix}.$$

Then we set

$$P_t = \begin{pmatrix} \mathbf{1}_n & 0 \\ 0 & \mathbf{1}_m X \end{pmatrix} \cdot \tilde{P}_t.$$

In this way the degrees of at most m rows are increased when passing from G_t to G_{t+1} (due to the restriction mentioned above $\tilde{P}_t$ does not increase the degrees), so the total number of zero rows in the coefficients is increased by n. Due to the restriction mentioned above the degrees of the rows of F_t and of G_t grow almost uniformly, i.e., they grow on average by $\frac{m}{m+n}$ when going from t to $t+1$.

After $t = \frac{d}{m} + \frac{d}{n} + O(1)$ steps the total number of zero rows in the coefficients of G_t is $\frac{m+n}{m}d + O(1)$ such that we can select m rows that produce at least d zero rows in the coefficients. These m rows form F and G.

We now consider unbalanced sequence lengths. Let ℓ_j be the length of sequence j, i.e., $a_{j,k}^{(i)}$ has been computed for all k and $0 \le i \le \ell_j$. Without loss of generality we can assume $\ell_1 \le \ell_2 \le \cdots \le \ell_n = \ell$. The sum of the lengths of all sequences has to satisfy again $\sum_j \ell_j \ge d \cdot (1 + \frac{n}{m}) + O(1)$. Moreover we can assume that $\ell_1 \ge \frac{d}{m}$, otherwise we could drop sequence 1 completely, thus facilitating our task. In this setting our goal is to achieve

$$FA \equiv G \pmod{X^{\ell + O(1)}}$$

with $d_F = \deg(F) \le \ell - \frac{d}{m}$, $\deg(G) \le \ell - \frac{d}{m}$ and

$$X^{\ell - \ell_k} \mid F_{\cdot,k} \qquad \text{(this denotes the } k\text{th column of } F\text{)}.$$

The latter condition will compensate our ignorance of some rows of the higher coefficients of A. Indeed, setting for simplicity $d_F = \ell - \frac{d}{m}$, the vectors

$$v_j = \sum_{k=1}^{n} \sum_{i=0}^{\ell_k - \frac{d}{m}} f_{j,k}^{(d_F - i)} \cdot M^i y_k$$

satisfy for $1 \le h \le m, 0 \le b < \frac{d}{m}$

$$(M^b x_h)^T v_j = \sum_{k=1}^{n} \sum_{i=0}^{\ell_k - \frac{d}{m}} f_{j,k}^{(d_F - i)} a_{k,h}^{(i+b)} = g_{j,h}^{(d_F + b)} = 0.$$

If $i+b > \ell_k$ (thus $a_{k,h}^{(i+b)}$ not being computed), we have $d_F - i < d_F + b - \ell_k \le \ell - \ell_k$, so $f_{j,k}^{(d_F - i)} = 0$ and the sum computes $g_{j,h}^{(d_F + b)}$.

Our new goal is achieved as before, but we will need ℓ steps and the construction of P_t has to be modified as follows. In step t we have $F_t A \equiv G_t + E_t X^t$

(mod X^{t+1}). Let $a \le n$ be maximal such that

$$\sum_{i=1}^{a-1}(m+i)(\ell_{n-i+1}-\ell_{n-i}) \le mt$$

(a will increase during the computation). In the Gaussian elimination of E_t we do not use the first $n-a$ rows for elimination. As a consequence, $\tilde{P}_t$ has the form

$$\tilde{P}_t = \begin{pmatrix} \mathbf{1}_{n-a} & * \\ 0 & * \end{pmatrix}.$$

Then we set

$$P_t = \begin{pmatrix} \mathbf{1}_{n-a}X & 0 & 0 \\ 0 & \mathbf{1}_a & 0 \\ 0 & 0 & \mathbf{1}_m X \end{pmatrix} \cdot \tilde{P}_t.$$

Therefore the sum of the degrees of F_t will be increased by $m+n-a$ and the number of zero rows in G_t will be increased by a when passing from t to $t+1$. For a fixed a, $\frac{(m+a)(\ell_{n-a+1}-\ell_{n-a})}{m}$ steps will increase the average degree of the last $m+a$ rows from $\ell-\ell_{n-a+1}$ to $\ell-\ell_{n-a}$. At this point a will be increased.

To see why $X^{\ell-\ell_k} \mid F_{\cdot,k}$ holds we have to describe the choice of F_{t_0} (and t_0). Let c be the number of maximal ℓ_j, i.e., $\ell_{n-c} < \ell_{n-c+1} = \ell_n$. Then F_{t_0} will be of the form

$$F_{t_0} = \begin{pmatrix} \mathbf{1}_{n-c}X^{t_0} & 0 \\ 0 & * \end{pmatrix}.$$

The last $m+c$ rows will be chosen such that they are of degree at most t_0-1 and such that the conditions in Coppersmith's algorithm are satisfied. This construction will lead to a value of t_0 near $\frac{m}{c}$ instead of the lower value near $\frac{m}{n}$ in the original algorithm.

Let k be such that $\ell_k < \ell$. As long as $n-a \ge k$ the kth column of F_t will have the only non-zero entry at row k and this will be X^t. Since $n-a \ge k$ holds for $t \le \ell-\ell_k$ this column will be divisible by $X^{\ell-\ell_k}$ for all $t \ge \ell-\ell_k$.

For RSA-768 we used the algorithm as described above in the subquadratic version of Thomé. The following variant might be useful in certain situations, e.g., if one of the sequences is much longer than the others.

If $\ell_{n-1} < \ell_n$, then for $t < \frac{(m+1)(\ell_n-\ell_{n-1})}{m}$ we have $a=1$ and P_t is of the form

$$P_t = \begin{pmatrix} \mathbf{1}_{n-1}X & * \\ 0 & * \end{pmatrix}.$$

A product of several of these P_t will have a similar form, namely an $(n-1)\times(n-1)$ unit matrix times a power of X in the upper left corner and zeros below it.

The basic operations in Thomé's subquadratic version are building a binary product tree of these P_t and doing truncated multiplications of intermediate products with $F_{t_0}A$ or similar polynomials. If we split the computation into two stages, first computing the product of all P_t for $t < \frac{(m+1)(\ell_n-\ell_{n-1})}{m}$ and then the remaining product, the matrix multiplications in the first stage become easier due to the special form of the P_t and its products.

Obviously this can be done in as many stages as there are different ℓ_j-values.